W3C Note

Simple Object Access Protocol (SOAP) 1.1

W3C Note 08 May 2000

This version:
http://www.w3.org/TR/2000/NOTE-SOAP-20000508
Latest version:
http://www.w3.org/TR/SOAP
Authors (alphabetically):

Don Box, DevelopMentor
David Ehnebuske, IBM
Gopal Kakivaya, Microsoft
Andrew Layman, Microsoft
Noah Mendelsohn, Lotus Development Corp.
Henrik Frystyk Nielsen, Microsoft
Satish Thatte, Microsoft
Dave Winer, UserLand Software, Inc.

Abstract

SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings defined in this document describe how to use SOAP in combination with HTTP and HTTP Extension Framework.

Status

This document is a submission to the World Wide Web Consortium (see Submission Request, W3C Staff Comment) to propose the formation of a working group in the area of XML-based protocols. Comments are welcome to the authors but you are encouraged to share your views on the W3C's public mailing list <xml-dist-app@w3.org> (see archives).

This document is a NOTE made available by the W3C for discussion only. Publication of this Note by W3C indicates no endorsement by W3C or the W3C Team, or any W3C Members. W3C has had no editorial control over the preparation of this Note. This document is a work in progress and may be updated, replaced, or rendered obsolete by other documents at any time.

A list of current W3C technical documents can be found at the Technical Reports page.

Table of Contents

1. Introduction

SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules. This allows SOAP to be used in a large variety of systems ranging from messaging systems to RPC.

SOAP consists of three parts:

- The SOAP envelope (see section 4) construct defines an overall framework for expressing **what** is in a message; **who** should deal with it, and **whether** it is optional or mandatory.
- The SOAP encoding rules (see section 5) defines a serialization mechanism that can be used to exchange instances of application-defined datatypes.
- The SOAP RPC representation (see section 7) defines a convention that can be used

to represent remote procedure calls and responses.

Although these parts are described together as part of SOAP, they are functionally orthogonal. In particular, the envelope and the encoding rules are defined in different namespaces in order to promote simplicity through modularity.

In addition to the SOAP envelope, the SOAP encoding rules and the SOAP RPC conventions, this specification defines two protocol bindings that describe how a SOAP message can be carried in HTTP [5] messages either with or without the HTTP Extension Framework [6].

1.1 Design Goals

A major design goal for SOAP is simplicity and extensibility. This means that there are several features from traditional messaging systems and distributed object systems that are not part of the core SOAP specification. Such features include

- Distributed garbage collection
- Boxcarring or batching of messages
- Objects-by-reference (which requires distributed garbage collection)
- Activation (which requires objects-by-reference)

1.2 Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [2].

The namespace prefixes "SOAP-ENV" and "SOAP-ENC" used in this document are associated with the SOAP namespaces "http://schemas.xmlsoap.org/soap/envelope/" and "http://schemas.xmlsoap.org/soap/encoding/" respectively.

Throughout this document, the namespace prefix "xsi" is assumed to be associated with the URI "http://www.w3.org/1999/XMLSchema-instance" which is defined in the XML Schemas specification [11]. Similarly, the namespace prefix "xsd" is assumed to be associated with the URI "http://www.w3.org/1999/XMLSchema" which is defined in [10]. The namespace prefix "tns" is used to indicate whatever is the target namespace of the current document. All other namespace prefixes are samples only.

Namespace URIs of the general form "some-URI" represent some application-dependent or

context-dependent URI [4].

This specification uses the augmented Backus-Naur Form (BNF) as described in RFC-2616 [5] for certain constructs.

1.3 Examples of SOAP Messages

In this example, a GetLastTradePrice SOAP request is sent to a StockQuote service. The request takes a string parameter, ticker symbol, and returns a float in the SOAP response. The SOAP Envelope element is the top element of the XML document representing the SOAP message. XML namespaces are used to disambiguate SOAP identifiers from application specific identifiers. The example illustrates the HTTP bindings defined in section 6. It is worth noting that the rules governing XML payload format in SOAP are entirely independent of the fact that the payload is carried in HTTP.

More examples are available in Appendix A.

Example 1 SOAP Message Embedded in HTTP Request

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Following is the response message containing the HTTP message with the SOAP message as the payload:

Example 2 SOAP Message Embedded in HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
```

```
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2. The SOAP Message Exchange Model

SOAP messages are fundamentally one-way transmissions from a sender to a receiver, but as illustrated above, SOAP messages are often combined to implement patterns such as request/response.

SOAP implementations can be optimized to exploit the unique characteristics of particular network systems. For example, the HTTP binding described in section 6 provides for SOAP response messages to be delivered as HTTP responses, using the same connection as the inbound request.

Regardless of the protocol to which SOAP is bound, messages are routed along a so-called "message path", which allows for processing at one or more intermediate nodes in addition to the ultimate destination.

A SOAP application receiving a SOAP message MUST process that message by performing the following actions in the order listed below:

1. Identify all parts of the SOAP message intended for that application (see section 4.2.2)
2. Verify that all mandatory parts identified in step 1 are supported by the application for this message (see section 4.2.3) and process them accordingly. If this is not the case then discard the message (see section 4.4). The processor MAY ignore optional parts identified in step 1 without affecting the outcome of the processing.
3. If the SOAP application is not the ultimate destination of the message then remove all parts identified in step 1 before forwarding the message.

Processing a message or a part of a message requires that the SOAP processor

understands, among other things, the exchange pattern being used (one way, request/response, multicast, etc.), the role of the recipient in that pattern, the employment (if any) of RPC mechanisms such as the one documented in section 7, the representation or encoding of data, as well as other semantics necessary for correct processing.

While attributes such as the SOAP encodingStyle attribute (see section 4.1.1) can be used to describe certain aspects of a message, this specification does not mandate a particular means by which the recipient makes such determinations in general. For example, certain applications will understand that a particular <getStockPrice> element signals an RPC request using the conventions of section 7, while another application may infer that all traffic directed to it is encoded as one way messages.

3. Relation to XML

All SOAP messages are encoded using XML (see [7] for more information on XML).

A SOAP application SHOULD include the proper SOAP namespace on all elements and attributes defined by SOAP in messages that it generates. A SOAP application MUST be able to process SOAP namespaces in messages that it receives. It MUST discard messages that have incorrect namespaces (see section 4.4) and it MAY process SOAP messages without SOAP namespaces as though they had the correct SOAP namespaces.

SOAP defines two namespaces (see [8] for more information on XML namespaces):

- The SOAP envelope has the namespace identifier "http://schemas.xmlsoap.org/soap/envelope/"
- The SOAP serialization has the namespace identifier "http://schemas.xmlsoap.org/soap/encoding/"

A SOAP message MUST NOT contain a Document Type Declaration. A SOAP message MUST NOT contain Processing Instructions. [7]

SOAP uses the local, unqualified "id" attribute of type "ID" to specify the unique identifier of an encoded element. SOAP uses the local, unqualified attribute "href" of type "uri-reference" to specify a reference to that value, in a manner conforming to the XML Specification [7], XML Schema Specification [11], and XML Linking Language Specification [9].

With the exception of the SOAP mustUnderstand attribute (see section 4.2.3) and the SOAP actor attribute (see section 4.2.2), it is generally permissible to have attributes and their values appear in XML instances or alternatively in schemas, with equal effect. That is, declaration in a DTD or schema with a default or fixed value is semantically equivalent to

appearance in an instance.

4. SOAP Envelope

A SOAP message is an XML document that consists of a mandatory SOAP envelope, an optional SOAP header, and a mandatory SOAP body. This XML document is referred to as a SOAP message for the rest of this specification. The namespace identifier for the elements and attributes defined in this section is "http://schemas.xmlsoap.org/soap/envelope/". A SOAP message contains the following:

- The Envelope is the top element of the XML document representing the message.
- The Header is a generic mechanism for adding features to a SOAP message in a decentralized manner without prior agreement between the communicating parties. SOAP defines a few attributes that can be used to indicate who should deal with a feature and whether it is optional or mandatory (see section 4.2)
- The Body is a container for mandatory information intended for the ultimate recipient of the message (see section 4.3). SOAP defines one element for the body, which is the Fault element used for reporting errors.

The grammar rules are as follows:

1. Envelope
    - The element name is "Envelope".
    - The element MUST be present in a SOAP message
    - The element MAY contain namespace declarations as well as additional attributes. If present, such additional attributes MUST be namespace-qualified. Similarly, the element MAY contain additional sub elements. If present these elements MUST be namespace-qualified and MUST follow the SOAP Body element.
2. Header (see section 4.2)
    - The element name is "Header".
    - The element MAY be present in a SOAP message. If present, the element MUST be the first immediate child element of a SOAP Envelope element.
    - The element MAY contain a set of header entries each being an immediate child element of the SOAP Header element. All immediate child elements of the SOAP Header element MUST be namespace-qualified.

3. Body (see section 4.3)
    - The element name is "Body".
    - The element MUST be present in a SOAP message and MUST be an immediate child element of a SOAP Envelope element. It MUST directly follow the SOAP Header element if present. Otherwise it MUST be the first immediate child element of the SOAP Envelope element.
    - The element MAY contain a set of body entries each being an immediate child element of the SOAP Body element. Immediate child elements of the SOAP Body element MAY be namespace-qualified. SOAP defines the SOAP Fault element, which is used to indicate error messages (see section 4.4).

4.1.1 SOAP encodingStyle Attribute

The SOAP encodingStyle global attribute can be used to indicate the serialization rules used in a SOAP message. This attribute MAY appear on any element, and is scoped to that element's contents and all child elements not themselves containing such an attribute, much as an XML namespace declaration is scoped. There is no default encoding defined for a SOAP message.

The attribute value is an ordered list of one or more URIs identifying the serialization rule or rules that can be used to deserialize the SOAP message indicated in the order of most specific to least specific. Examples of values are

```
"http://schemas.xmlsoap.org/soap/encoding/"
"http://my.host/encoding/restricted http://my.host/encoding/"
""
```

The serialization rules defined by SOAP in section 5 are identified by the URI "http://schemas.xmlsoap.org/soap/encoding/". Messages using this particular serialization SHOULD indicate this using the SOAP encodingStyle attribute. In addition, all URIs syntactically beginning with "http://schemas.xmlsoap.org/soap/encoding/" indicate conformance with the SOAP encoding rules defined in section 5 (though with potentially tighter rules added).

A value of the zero-length URI ("") explicitly indicates that no claims are made for the encoding style of contained elements. This can be used to turn off any claims from containing elements.

4.1.2 Envelope Versioning Model

SOAP does not define a traditional versioning model based on major and minor version

numbers. A SOAP message MUST have an Envelope element associated with the "http://schemas.xmlsoap.org/soap/envelope/" namespace. If a message is received by a SOAP application in which the SOAP Envelope element is associated with a different namespace, the application MUST treat this as a version error and discard the message. If the message is received through a request/response protocol such as HTTP, the application MUST respond with a SOAP VersionMismatch faultcode message (see section 4.4) using the SOAP "http://schemas.xmlsoap.org/soap/envelope/" namespace.

4.2 SOAP Header

SOAP provides a flexible mechanism for extending a message in a decentralized and modular way without prior knowledge between the communicating parties. Typical examples of extensions that can be implemented as header entries are authentication, transaction management, payment etc.

The Header element is encoded as the first immediate child element of the SOAP Envelope XML element. All immediate child elements of the Header element are called header entries.

The encoding rules for header entries are as follows:

1. A header entry is identified by its fully qualified element name, which consists of the namespace URI and the local name. All immediate child elements of the SOAP Header element MUST be namespace-qualified.
2. The SOAP encodingStyle attribute MAY be used to indicate the encoding style used for the header entries (see section 4.1.1).
3. The SOAP mustUnderstand attribute (see section 4.2.3) and SOAP actor attribute (see section 4.2.2) MAY be used to indicate how to process the entry and by whom (see section 4.2.1).

4.2.1 Use of Header Attributes

The SOAP Header attributes defined in this section determine how a recipient of a SOAP message should process the message as described in section 2. A SOAP application generating a SOAP message SHOULD only use the SOAP Header attributes on immediate child elements of the SOAP Header element. The recipient of a SOAP message MUST ignore all SOAP Header attributes that are not applied to an immediate child element of the SOAP Header element.

An example is a header with an element identifier of "Transaction", a "mustUnderstand" value of "1", and a value of 5. This would be encoded as follows:

```
<SOAP-ENV:Header>
   <t:Transaction
      xmlns:t="some-URI" SOAP-ENV:mustUnderstand="1">
          5
   </t:Transaction>
</SOAP-ENV:Header>
```

4.2.2 SOAP actor Attribute

A SOAP message travels from the originator to the ultimate destination, potentially by passing through a set of SOAP intermediaries along the message path. A SOAP intermediary is an application that is capable of both receiving and forwarding SOAP messages. Both intermediaries as well as the ultimate destination are identified by a URI.

Not all parts of a SOAP message may be intended for the ultimate destination of the SOAP message but, instead, may be intended for one or more of the intermediaries on the message path. The role of a recipient of a header element is similar to that of accepting a contract in that it cannot be extended beyond the recipient. That is, a recipient receiving a header element MUST NOT forward that header element to the next application in the SOAP message path. The recipient MAY insert a similar header element but in that case, the contract is between that application and the recipient of that header element.

The SOAP actor global attribute can be used to indicate the recipient of a header element. The value of the SOAP actor attribute is a URI. The special URI "http://schemas.xmlsoap.org/soap/actor/next" indicates that the header element is intended for the very first SOAP application that processes the message. This is similar to the hop-by-hop scope model represented by the Connection header field in HTTP.

Omitting the SOAP actor attribute indicates that the recipient is the ultimate destination of the SOAP message.

This attribute MUST appear in the SOAP message instance in order to be effective (see section 3 and 4.2.1).

4.2.3 SOAP mustUnderstand Attribute

The SOAP mustUnderstand global attribute can be used to indicate whether a header entry is mandatory or optional for the recipient to process. The recipient of a header entry is defined by the SOAP actor attribute (see section 4.2.2). The value of the mustUnderstand attribute is either "1" or "0". The absence of the SOAP mustUnderstand attribute is semantically equivalent to its presence with the value "0".

If a header element is tagged with a SOAP mustUnderstand attribute with a value of "1", the

recipient of that header entry either MUST obey the semantics (as conveyed by the fully qualified name of the element) and process correctly to those semantics, or MUST fail processing the message (see section 4.4).

The SOAP mustUnderstand attribute allows for robust evolution. Elements tagged with the SOAP mustUnderstand attribute with a value of "1" MUST be presumed to somehow modify the semantics of their parent or peer elements. Tagging elements in this manner assures that this change in semantics will not be silently (and, presumably, erroneously) ignored by those who may not fully understand it.

This attribute MUST appear in the instance in order to be effective (see section 3 and 4.2.1).

4.3 SOAP Body

The SOAP Body element provides a simple mechanism for exchanging mandatory information intended for the ultimate recipient of the message. Typical uses of the Body element include marshalling RPC calls and error reporting.

The Body element is encoded as an immediate child element of the SOAP Envelope XML element. If a Header element is present then the Body element MUST immediately follow the Header element, otherwise it MUST be the first immediate child element of the Envelope element.

All immediate child elements of the Body element are called body entries and each body entry is encoded as an independent element within the SOAP Body element.

The encoding rules for body entries are as follows:

1. A body entry is identified by its fully qualified element name, which consists of the namespace URI and the local name. Immediate child elements of the SOAP Body element MAY be namespace-qualified.
2. The SOAP encodingStyle attribute MAY be used to indicate the encoding style used for the body entries (see section 4.1.1).

SOAP defines one body entry, which is the Fault entry used for reporting errors (see section 4.4).

4.3.1 Relationship between SOAP Header and Body

While the Header and Body are defined as independent elements, they are in fact related. The relationship between a body entry and a header entry is as follows: A body entry is semantically equivalent to a header entry intended for the default actor and with a SOAP

mustUnderstand attribute with a value of "1". The default actor is indicated by not using the actor attribute (see section 4.2.2).

4.4 SOAP Fault

The SOAP Fault element is used to carry error and/or status information within a SOAP message. If present, the SOAP Fault element MUST appear as a body entry and MUST NOT appear more than once within a Body element.

The SOAP Fault element defines the following four subelements:

faultcode

The faultcode element is intended for use by software to provide an algorithmic mechanism for identifying the fault. The faultcode MUST be present in a SOAP Fault element and the faultcode value MUST be a qualified name as defined in [8], section 3. SOAP defines a small set of SOAP fault codes covering basic SOAP faults (see section 4.4.1)

faultstring

The faultstring element is intended to provide a human readable explanation of the fault and is not intended for algorithmic processing. The faultstring element is similar to the 'Reason-Phrase' defined by HTTP (see [5], section 6.1). It MUST be present in a SOAP Fault element and SHOULD provide at least some information explaining the nature of the fault.

faultactor

The faultactor element is intended to provide information about who caused the fault to happen within the message path (see section 2). It is similar to the SOAP actor attribute (see section 4.2.2) but instead of indicating the destination of the header entry, it indicates the source of the fault. The value of the faultactor attribute is a URI identifying the source. Applications that do not act as the ultimate destination of the SOAP message MUST include the faultactor element in a SOAP Fault element. The ultimate destination of a message MAY use the faultactor element to indicate explicitly that it generated the fault (see also the detail element below).

detail

The detail element is intended for carrying application specific error information related to the Body element. It MUST be present if the contents of the Body element could not be successfully processed. It MUST NOT be used to carry information about error information belonging to header entries. Detailed error information belonging to header entries MUST be carried within header entries.

The absence of the detail element in the Fault element indicates that the fault is not related to processing of the Body element. This can be used to distinguish whether the Body element was processed or not in case of a fault situation.

All immediate child elements of the detail element are called detail entries and each detail entry is encoded as an independent element within the detail element.

The encoding rules for detail entries are as follows (see also example 10):

1. A detail entry is identified by its fully qualified element name, which consists of the namespace URI and the local name. Immediate child elements of the detail element MAY be namespace-qualified.
2. The SOAP encodingStyle attribute MAY be used to indicate the encoding style used for the detail entries (see section 4.1.1).

Other Fault subelements MAY be present, provided they are namespace-qualified.

4.4.1 SOAP Fault Codes

The faultcode values defined in this section MUST be used in the faultcode element when describing faults defined by this specification. The namespace identifier for these faultcode values is "http://schemas.xmlsoap.org/soap/envelope/". Use of this space is recommended (but not required) in the specification of methods defined outside of the present specification.

The default SOAP faultcode values are defined in an extensible manner that allows for new SOAP faultcode values to be defined while maintaining backwards compatibility with existing faultcode values. The mechanism used is very similar to the 1xx, 2xx, 3xx etc basic status classes classes defined in HTTP (see [5] section 10). However, instead of integers, they are defined as XML qualified names (see [8] section 3). The character "." (dot) is used as a separator of faultcode values indicating that what is to the left of the dot is a more generic fault code value than the value to the right. Example

```
Client.Authentication
```

The set of faultcode values defined in this document is:

Name	Meaning
VersionMismatch	The processing party found an invalid namespace for the SOAP Envelope element (see section 4.1.2)
MustUnderstand	An immediate child element of the SOAP Header element that was either not understood or not obeyed by the processing party contained a SOAP mustUnderstand attribute with a value of "1" (see section 4.2.3)

Client	The Client class of errors indicate that the message was incorrectly formed or did not contain the appropriate information in order to succeed. For example, the message could lack the proper authentication or payment information. It is generally an indication that the message should not be resent without change. See also section 4.4 for a description of the SOAP Fault detail sub-element.
Server	The Server class of errors indicate that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to the processing of the message. For example, processing could include communicating with an upstream processor, which didn't respond. The message may succeed at a later point in time. See also section 4.4 for a description of the SOAP Fault detail sub-element.

5. SOAP Encoding

The SOAP encoding style is based on a simple type system that is a generalization of the common features found in type systems in programming languages, databases and semi-structured data. A type either is a simple (scalar) type or is a compound type constructed as a composite of several parts, each with a type. This is described in more detail below. This section defines rules for serialization of a graph of typed objects. It operates on two levels. First, given a schema in any notation consistent with the type system described, a schema for an XML grammar may be constructed. Second, given a type-system schema and a particular graph of values conforming to that schema, an XML instance may be constructed. In reverse, given an XML instance produced in accordance with these rules, and given also the original schema, a copy of the original value graph may be constructed.

The namespace identifier for the elements and attributes defined in this section is "http://schemas.xmlsoap.org/soap/encoding/". The encoding samples shown assume all namespace declarations are at a higher element level.

Use of the data model and encoding style described in this section is encouraged but not required; other data models and encodings can be used in conjunction with SOAP (see section 4.1.1).

5.1 Rules for Encoding Types in XML

XML allows very flexible encoding of data. SOAP defines a narrower set of rules for encoding. This section defines the encoding rules at a high level, and the next section describes the encoding rules for specific types when they require more detail. The encodings described in this section can be used in conjunction with the mapping of RPC

calls and responses specified in Section 7.

To describe encoding, the following terminology is used:

1. A "value" is a string, the name of a measurement (number, date, enumeration, etc.) or a composite of several such primitive values. All values are of specific types.
2. A "simple value" is one without named parts. Examples of simple values are particular strings, integers, enumerated values etc.
3. A "compound value" is an aggregate of relations to other values. Examples of Compound Values are particular purchase orders, stock reports, street addresses, etc.
4. Within a compound value, each related value is potentially distinguished by a role name, ordinal or both. This is called its "accessor." Examples of compound values include particular Purchase Orders, Stock Reports etc. Arrays are also compound values. It is possible to have compound values with several accessors each named the same, as for example, RDF does.
5. An "array" is a compound value in which ordinal position serves as the only distinction among member values.
6. A "struct" is a compound value in which accessor name is the only distinction among member values, and no accessor has the same name as any other.
7. A "simple type" is a class of simple values. Examples of simple types are the classes called "string," "integer," enumeration classes, etc.
8. A "compound type" is a class of compound values. An example of a compound type is the class of purchase order values sharing the same accessors (shipTo, totalCost, etc.) though with potentially different values (and perhaps further constrained by limits on certain values).
9. Within a compound type, if an accessor has a name that is distinct within that type but is not distinct with respect to other types, that is, the name plus the type together are needed to make a unique identification, the name is called "locally scoped." If however the name is based in part on a Uniform Resource Identifier, directly or indirectly, such that the name alone is sufficient to uniquely identify the accessor irrespective of the type within which it appears, the name is called "universally scoped."
10. Given the information in the schema relative to which a graph of values is serialized, it is possible to determine that some values can only be related by a single instance of an accessor. For others, it is not possible to make this determination. If only one accessor can reference it, a value is considered "single-reference". If referenced by more than one, actually or potentially, it is "multi-reference." Note that it is possible for a certain value to be considered "single-reference" relative to one schema and "multi-

reference" relative to another.

11. Syntactically, an element may be "independent" or "embedded." An independent element is any element appearing at the top level of a serialization. All others are embedded elements.

Although it is possible to use the xsi:type attribute such that a graph of values is self-describing both in its structure and the types of its values, the serialization rules permit that the types of values MAY be determinate only by reference to a schema. Such schemas MAY be in the notation described by "XML Schema Part 1: Structures" [10] and "XML Schema Part 2: Datatypes" [11] or MAY be in any other notation. Note also that, while the serialization rules apply to compound types other than arrays and structs, many schemas will contain only struct and array types.

The rules for serialization are as follows:

1. All values are represented as element content. A multi-reference value MUST be represented as the content of an independent element. A single-reference value SHOULD not be (but MAY be).
2. For each element containing a value, the type of the value MUST be represented by at least one of the following conditions: (a) the containing element instance contains an xsi:type attribute, (b) the containing element instance is itself contained within an element containing a (possibly defaulted) SOAP-ENC:arrayType attribute or (c) or the name of the element bears a definite relation to the type, that type then determinable from a schema.
3. A simple value is represented as character data, that is, without any subelements. Every simple value must have a type that is either listed in the XML Schemas Specification, part 2 [11] or whose source type is listed therein (see also section 5.2).
4. A Compound Value is encoded as a sequence of elements, each accessor represented by an embedded element whose name corresponds to the name of the accessor. Accessors whose names are local to their containing types have unqualified element names; all others have qualified names (see also section 5.4).
5. A multi-reference simple or compound value is encoded as an independent element containing a local, unqualified attribute named "id" and of type "ID" per the XML Specification [7]. Each accessor to this value is an empty element having a local, unqualified attribute named "href" and of type "uri-reference" per the XML Schema Specification [11], with a "href" attribute value of a URI fragment identifier referencing the corresponding independent element.
6. Strings and byte arrays are represented as multi-reference simple types, but special rules allow them to be represented efficiently for common cases (see also section 5.2.1 and 5.2.3). An accessor to a string or byte-array value MAY have an attribute

named "id" and of type "ID" per the XML Specification [7]. If so, all other accessors to the same value are encoded as empty elements having a local, unqualified attribute named "href" and of type "uri-reference" per the XML Schema Specification [11], with a "href" attribute value of a URI fragment identifier referencing the single element containing the value.

7. It is permissible to encode several references to a value as though these were references to several distinct values, but only when from context it is known that the meaning of the XML instance is unaltered.

8. Arrays are compound values (see also section 5.4.2). SOAP arrays are defined as having a type of "SOAP-ENC:Array" or a type derived there from.

   SOAP arrays have one or more dimensions (rank) whose members are distinguished by ordinal position. An array value is represented as a series of elements reflecting the array, with members appearing in ascending ordinal sequence. For multi-dimensional arrays the dimension on the right side varies most rapidly. Each member element is named as an independent element (see rule 2).

   SOAP arrays can be single-reference or multi-reference values, and consequently may be represented as the content of either an embedded or independent element.

   SOAP arrays MUST contain a "SOAP-ENC:arrayType" attribute whose value specifies the type of the contained elements as well as the dimension(s) of the array. The value of the "SOAP-ENC:arrayType" attribute is defined as follows:

```
arrayTypeValue = atype asize
atype          = QName *( rank )
rank           = "[" *( "," ) "]"
asize          = "[" #length "]"
length         = 1*DIGIT
```

   The "atype"construct is the type name of the contained elements expressed as a QName as would appear in the "type"attribute of an XML Schema element declaration and acts as a type constraint (meaning that all values of contained elements are asserted to conform to the indicated type; that is, the type cited in SOAP-ENC:arrayType must be the type or a supertype of every array member). In the case of arrays of arrays or "jagged arrays", the type component is encoded as the "innermost"type name followed by a rank construct for each level of nested arrays starting from 1. Multi-dimensional arrays are encoded using a comma for each dimension starting from 1.

   The "asize"construct contains a comma separated list of zero, one, or more integers indicating the lengths of each dimension of the array. A value of zero integers indicates that no particular quantity is asserted but that the size may be determined by inspection of the actual members.

For example, an array with 5 members of type array of integers would have an arrayTypeValue value of "int[][5]"of which the atype value is "int[]"and the asize value is "[5]". Likewise, an array with 3 members of type two-dimensional arrays of integers would have an arrayTypeValue value of "int[,][3]"of which the atype value is "int[,]"and the asize value is "[3]".

A SOAP array member MAY contain a "SOAP-ENC:offset"attribute indicating the offset position of that item in the enclosing array. This can be used to indicate the offset position of a partially represented array (see section 5.4.2.1). Likewise, an array member MAY contain a "SOAP-ENC:position"attribute indicating the position of that item in the enclosing array. This can be used to describe members of sparse arrays (see section 5.4.2.2). The value of the "SOAP-ENC:offset"and the "SOAP-ENC:position"attribute is defined as follows:

```
arrayPoint = "[" #length "]"
```

with offsets and positions based at 0.

9. A NULL value or a default value MAY be represented by omission of the accessor element. A NULL value MAY also be indicated by an accessor element containing the attribute xsi:null with value '1' or possibly other application-dependent attributes and values.

Note that rule 2 allows independent elements and also elements representing the members of arrays to have names which are not identical to the type of the contained value.

5.2 Simple Types

For simple types, SOAP adopts all the types found in the section "Built-in datatypes" of the "XML Schema Part 2: Datatypes" Specification [11], both the value and lexical spaces. Examples include:

Type	Example
int	58502
float	314159265358979E+1
negativeInteger	-32768
string	Louis "Satchmo" Armstrong

The datatypes declared in the XML Schema specification may be used directly in element

schemas. Types derived from these may also be used. An example of a schema fragment and corresponding instance data with elements of these types is:

```
<element name="age" type="int"/>
<element name="height" type="float"/>
<element name="displacement" type="negativeInteger"/>
<element name="color">
  <simpleType base="xsd:string">
    <enumeration value="Green"/>
    <enumeration value="Blue"/>
  </simpleType>
</element>

<age>45</age>
<height>5.9</height>
<displacement>-450</displacement>
<color>Blue</color>
```

All simple values MUST be encoded as the content of elements whose type is either defined in "XML Schema Part 2: Datatypes" Specification [11], or is based on a type found there by using the mechanisms provided in the XML Schema specification.

If a simple value is encoded as an independent element or member of a heterogenous array it is convenient to have an element declaration corresponding to the datatype. Because the "XML Schema Part 2: Datatypes" Specification [11] includes type definitions but does not include corresponding element declarations, the SOAP-ENC schema and namespace declares an element for every simple datatype. These MAY be used.

```
<SOAP-ENC:int id="int1">45</SOAP-ENC:int>
```

5.2.1 Strings

The datatype "string" is defined in "XML Schema Part 2: Datatypes" Specification [11]. Note that this is not identical to the type called "string" in many database or programming languages, and in particular may forbid some characters those languages would permit. (Those values must be represented by using some datatype other than xsd:string.)

A string MAY be encoded as a single-reference or a multi-reference value.

The containing element of the string value MAY have an "id" attribute. Additional accessor elements MAY then have matching "href" attributes.

For example, two accessors to the same string could appear, as follows:

```
<greeting id="String-0">Hello</greeting>
<salutation href="#String-0"/>
```

However, if the fact that both accessors reference the same instance of the string (or subtype of string) is immaterial, they may be encoded as two single-reference values as follows:

```
<greeting>Hello</greeting>
<salutation>Hello</salutation>
```

Schema fragments for these examples could appear similar to the following:

```
<element name="greeting" type="SOAP-ENC:string"/>
<element name="salutation" type="SOAP-ENC:string"/>
```

(In this example, the type SOAP-ENC:string is used as the element's type as a convenient way to declare an element whose datatype is "xsd:string" and which also allows an "id" and "href" attribute. See the SOAP Encoding schema for the exact definition. Schemas MAY use these declarations from the SOAP Encoding schema but are not required to.)

5.2.2 Enumerations

The "XML Schema Part 2: Datatypes" Specification [11] defines a mechanism called "enumeration." The SOAP data model adopts this mechanism directly. However, because programming and other languages often define enumeration somewhat differently, we spell-out the concept in more detail here and describe how a value that is a member of an enumerated list of possible values is to be encoded. Specifically, it is encoded as the name of the value.

"Enumeration" as a concept indicates a set of distinct names. A specific enumeration is a specific list of distinct values appropriate to the base type. For example the set of color names ("Green", "Blue", "Brown") could be defined as an enumeration based on the string built-in type. The values ("1", "3", "5") are a possible enumeration based on integer, and so on. "XML Schema Part 2: Datatypes" [11] supports enumerations for all of the simple types except for boolean. The language of "XML Schema Part 1: Structures" Specification [10] can be used to define enumeration types. If a schema is generated from another notation in which no specific base type is applicable, use "string". In the following schema example "EyeColor" is defined as a string with the possible values of "Green", "Blue", or "Brown" enumerated, and instance data is shown accordingly.

```
<element name="EyeColor" type="tns:EyeColor"/>
<simpleType name="EyeColor" base="xsd:string">
   <enumeration value="Green"/>
   <enumeration value="Blue"/>
```

```
   <enumeration value="Brown"/>
</simpleType>
```

```
<Person>
   <Name>Henry Ford</Name>
   <Age>32</Age>
   <EyeColor>Brown</EyeColor>
</Person>
```

5.2.3 Array of Bytes

An array of bytes MAY be encoded as a single-reference or a multi-reference value. The rules for an array of bytes are similar to those for a string.

In particular, the containing element of the array of bytes value MAY have an "id" attribute. Additional accessor elements MAY then have matching "href" attributes.

The recommended representation of an opaque array of bytes is the 'base64' encoding defined in XML Schemas [10][11], which uses the base64 encoding algorithm defined in 2045 [13]. However, the line length restrictions that normally apply to base64 data in MIME do not apply in SOAP. A "SOAP-ENC:base64" subtype is supplied for use with SOAP.

```
<picture xsi:type="SOAP-ENC:base64">
   aG93IG5vDyBicm73biBjb3cNCg==
</picture>
```

5.3 Polymorphic Accessor

Many languages allow accessors that can polymorphically access values of several types, each type being available at run time. A polymorphic accessor instance MUST contain an "xsi:type" attribute that describes the type of the actual value.

For example, a polymorphic accessor named "cost" with a value of type "xsd:float" would be encoded as follows:

```
<cost xsi:type="xsd:float">29.95</cost>
```

as contrasted with a cost accessor whose value's type is invariant, as follows:

```
<cost>29.95</cost>
```

5.4 Compound types

SOAP defines types corresponding to the following structural patterns often found in programming languages:

Struct
A "struct" is a compound value in which accessor name is the only distinction among member values, and no accessor has the same name as any other.

Array
An "array" is a compound value in which ordinal position serves as the only distinction among member values.

SOAP also permits serialization of data that is neither a Struct nor an Array, for example data such as is found in a Directed-Labeled-Graph Data Model in which a single node has many distinct accessors, some of which occur more than once. SOAP serialization does not require that the underlying data model make an ordering distinction among accessors, but if such an order exists, the accessors MUST be encoded in that sequence.

5.4.1 Compound Values, Structs and References to Values

The members of a Compound Value are encoded as accessor elements. When accessors are distinguished by their name (as for example in a struct), the accessor name is used as the element name. Accessors whose names are local to their containing types have unqualified element names; all others have qualified names.

The following is an example of a struct of type "Book":

```
<e:Book>
   <author>Henry Ford</author>
   <preface>Prefatory text</preface>
   <intro>This is a book.</intro>
</e:Book>
```

And this is a schema fragment describing the above structure:

```
<element name="Book">
<complexType>
  <element name="author" type="xsd:string"/>
  <element name="preface" type="xsd:string"/>
   <element name="intro" type="xsd:string"/>
</complexType>
</e:Book>
```

Below is an example of a type with both simple and complex members. It shows two levels of referencing. Note that the "href" attribute of the "Author" accessor element is a reference

to the value whose "id" attribute matches. A similar construction appears for the "Address".

```
<e:Book>
   <title>My Life and Work</title>
   <author href="#Person-1"/>
</e:Book>
<e:Person id="Person-1">
   <name>Henry Ford</name>
   <address href="#Address-2"/>
</e:Person>
<e:Address id="Address-2">
   <email>mailto:henryford@hotmail.com</email>
   <web>http://www.henryford.com</web>
</e:Address>
```

The form above is appropriate when the "Person" value and the "Address" value are multi-reference. If these were instead both single-reference, they SHOULD be embedded, as follows:

```
<e:Book>
   <title>My Life and Work</title>
   <author>
       <name>Henry Ford</name>
       <address>
          <email>mailto:henryford@hotmail.com</email>
          <web>http://www.henryford.com</web>
       </address>
   </author>
</e:Book>
```

If instead there existed a restriction that no two persons can have the same address in a given instance and that an address can be either a Street-address or an Electronic-address, a Book with two authors would be encoded as follows:

```
<e:Book>
   <title>My Life and Work</title>
   <firstauthor href="#Person-1"/>
   <secondauthor href="#Person-2"/>
</e:Book>
<e:Person id="Person-1">
   <name>Henry Ford</name>
   <address xsi:type="m:Electronic-address">
       <email>mailto:henryford@hotmail.com</email>
       <web>http://www.henryford.com</web>
   </address>
```

```
</e:Person>
<e:Person id="Person-2">
   <name>Samuel Crowther</name>
   <address xsi:type="n:Street-address">
       <street>Martin Luther King Rd</street>
       <city>Raleigh</city>
       <state>North Carolina</state>
   </address>
</e:Person>
```

Serializations can contain references to values not in the same resource:

```
<e:Book>
   <title>Paradise Lost</title>
   <firstauthor href="http://www.dartmouth.edu/~milton/"/>
</e:Book>
```

And this is a schema fragment describing the above structures:

```
<element name="Book" type="tns:Book"/>
<complexType name="Book">
   <!-- Either the following group must occur or else the
        href attribute must appear, but not both. -->
   <sequence minOccurs="0" maxOccurs="1">
       <element name="title" type="xsd:string"/>
       <element name="firstauthor" type="tns:Person"/>
       <element name="secondauthor" type="tns:Person"/>
   </sequence>
   <attribute name="href" type="uriReference"/>
   <attribute name="id" type="ID"/>
   <anyAttribute namespace="##other"/>
</complexType>

<element name="Person" base="tns:Person"/>
<complexType name="Person">
   <!-- Either the following group must occur or else the
        href attribute must appear, but not both. -->
   <sequence minOccurs="0" maxOccurs="1">
       <element name="name" type="xsd:string"/>
       <element name="address" type="tns:Address"/>
   </sequence>
   <attribute name="href" type="uriReference"/>
   <attribute name="id" type="ID"/>
   <anyAttribute namespace="##other"/>
</complexType>
```

```
<element name="Address" base="tns:Address"/>
<complexType name="Address">
   <!-- Either the following group must occur or else the
        href attribute must appear, but not both. -->
   <sequence minOccurs="0" maxOccurs="1">
       <element name="street" type="xsd:string"/>
       <element name="city" type="xsd:string"/>
       <element name="state" type="xsd:string"/>
   </sequence>
   <attribute name="href" type="uriReference"/>
   <attribute name="id" type="ID"/>
   <anyAttribute namespace="##other"/>
</complexType>
```

5.4.2 Arrays

SOAP arrays are defined as having a type of "SOAP-ENC:Array" or a type derived there from (see also rule 8). Arrays are represented as element values, with no specific constraint on the name of the containing element (just as values generally do not constrain the name of their containing element).

Arrays can contain elements which themselves can be of any type, including nested arrays. New types formed by restrictions of SOAP-ENC:Array can also be created to represent, for example, arrays limited to integers or arrays of some user-defined enumeration.

The representation of the value of an array is an ordered sequence of elements constituting the items of the array. Within an array value, element names are not significant for distinguishing accessors. Elements may have any name. In practice, elements will frequently be named so that their declaration in a schema suggests or determines their type. As with compound types generally, if the value of an item in the array is a single-reference value, the item contains its value. Otherwise, the item references its value via an "href" attribute.

The following example is a schema fragment and an array containing integer array members.

```
<element name="myFavoriteNumbers"
        type="SOAP-ENC:Array"/>

<myFavoriteNumbers
  SOAP-ENC:arrayType="xsd:int[2]">
   <number>3</number>
   <number>4</number>
</myFavoriteNumbers>
```

In that example, the array "myFavoriteNumbers" contains several members each of which is a value of type SOAP-ENC:int. This can be determined by inspection of the SOAP-ENC:arrayType attribute. Note that the SOAP-ENC:Array type allows unqualified element names without restriction. These convey no type information, so when used they must either have an xsi:type attribute or the containing element must have a SOAP-ENC:arrayType attribute. Naturally, types derived from SOAP-ENC:Array may declare local elements, with type information.

As previously noted, the SOAP-ENC schema contains declarations of elements with names corresponding to each simple type in the "XML Schema Part 2: Datatypes" Specification [11]. It also contains a declaration for "Array". Using these, we might write

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[2]">
   <SOAP-ENC:int>3</SOAP-ENC:int>
   <SOAP-ENC:int>4</SOAP-ENC:int>
</SOAP-ENC:Array>
```

Arrays can contain instances of any subtype of the specified arrayType. That is, the members may be of any type that is substitutable for the type specified in the arrayType attribute, according to whatever substitutability rules are expressed in the schema. So, for example, an array of integers can contain any type derived from integer (for example "int" or any user-defined derivation of integer). Similarly, an array of "address" might contain a restricted or extended type such as "internationalAddress". Because the supplied SOAP-ENC:Array type admits members of any type, arbitrary mixtures of types can be contained unless specifically limited by use of the arrayType attribute.

Types of member elements can be specified using the xsi:type attribute in the instance, or by declarations in the schema of the member elements, as the following two arrays demonstrate respectively.

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:ur-type[4]">
   <thing xsi:type="xsd:int">12345</thing>
   <thing xsi:type="xsd:decimal">6.789</thing>
   <thing xsi:type="xsd:string">
      Of Mans First Disobedience, and the Fruit
      Of that Forbidden Tree, whose mortal tast
      Brought Death into the World, and all our woe,
   </thing>
   <thing xsi:type="xsd:uriReference">
      http://www.dartmouth.edu/~milton/reading_room/
   </thing>
</SOAP-ENC:Array>
```

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:ur-type[4]">
```

```
   <SOAP-ENC:int>12345</SOAP-ENC:int>
   <SOAP-ENC:decimal>6.789</SOAP-ENC:decimal>
   <xsd:string>
      Of Mans First Disobedience, and the Fruit
      Of that Forbidden Tree, whose mortal tast
      Brought Death into the World, and all our woe,
   </xsd:string>
   <SOAP-ENC:uriReference>
      http://www.dartmouth.edu/~milton/reading_room/
   </SOAP-ENC:uriReference >
</SOAP-ENC:Array>
```

Array values may be structs or other compound values. For example an array of "xyz:Order" structs :

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xyz:Order[2]">
   <Order>
       <Product>Apple</Product>
       <Price>1.56</Price>
   </Order>
   <Order>
       <Product>Peach</Product>
       <Price>1.48</Price>
   </Order>
</SOAP-ENC:Array>
```

Arrays may have other arrays as member values. The following is an example of an array of two arrays, each of which is an array of strings.

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[][2]">
   <item href="#array-1"/>
   <item href="#array-2"/>
</SOAP-ENC:Array>
<SOAP-ENC:Array id="array-1" SOAP-
ENC:arrayType="xsd:string[2]">
   <item>r1c1</item>
   <item>r1c2</item>
   <item>r1c3</item>
</SOAP-ENC:Array>
<SOAP-ENC:Array id="array-2" SOAP-
ENC:arrayType="xsd:string[2]">
   <item>r2c1</item>
   <item>r2c2</item>
</SOAP-ENC:Array>
```

The element containing an array value does not need to be named "SOAP-ENC:Array". It may have any name, provided that the type of the element is either SOAP-ENC:Array or is derived from SOAP-ENC:Array by restriction. For example, the following is a fragment of a schema and a conforming instance array.

```
<simpleType name="phoneNumber" base="string"/>

<element name="ArrayOfPhoneNumbers">
  <complexType base="SOAP-ENC:Array">
    <element name="phoneNumber" type="tns:phoneNumber"
maxOccurs="unbounded"/>
  </complexType>
  <anyAttribute/>
</element>
```

```
<xyz:ArrayOfPhoneNumbers SOAP-
ENC:arrayType="xyz:phoneNumber[2]">
   <phoneNumber>206-555-1212</phoneNumber>
   <phoneNumber>1-888-123-4567</phoneNumber>
</xyz:ArrayOfPhoneNumbers>
```

Arrays may be multi-dimensional. In this case, more than one size will appear within the asize part of the arrayType attribute:

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[2,3]">
   <item>r1c1</item>
   <item>r1c2</item>
   <item>r1c3</item>
   <item>r2c1</item>
   <item>r2c2</item>
   <item>r2c3</item>
</SOAP-ENC:Array>
```

While the examples above have shown arrays encoded as independent elements, array values MAY also appear embedded and SHOULD do so when they are known to be single reference.

The following is an example of a schema fragment and an array of phone numbers embedded in a struct of type "Person" and accessed through the accessor "phone-numbers":

```
<simpleType name="phoneNumber" base="string"/>

<element name="ArrayOfPhoneNumbers">
  <complexType base="SOAP-ENC:Array">
```

```
    <element name="phoneNumber" type="tns:phoneNumber"
maxOccurs="unbounded"/>
  </complexType>
  <anyAttribute/>
</element>

<element name="Person">
  <complexType>
    <element name="name" type="string"/>
    <element name="phoneNumbers"
type="tns:ArrayOfPhoneNumbers"/>
  </complexType>
</element>
```

```
<xyz:Person>
   <name>John Hancock</name>
   <phoneNumbers SOAP-ENC:arrayType="xyz:phoneNumber[2]">
       <phoneNumber>206-555-1212</phoneNumber>
       <phoneNumber>1-888-123-4567</phoneNumber>
   </phoneNumbers>
</xyz:Person>
```

Here is another example of a single-reference array value encoded as an embedded element whose containing element name is the accessor name:

```
<xyz:PurchaseOrder>
   <CustomerName>Henry Ford</CustomerName>
   <ShipTo>
       <Street>5th Ave</Street>
       <City>New York</City>
       <State>NY</State>
       <Zip>10010</Zip>
   </ShipTo>
   <PurchaseLineItems SOAP-ENC:arrayType="Order[2]">
       <Order>
           <Product>Apple</Product>
           <Price>1.56</Price>
       </Order>
       <Order>
           <Product>Peach</Product>
           <Price>1.48</Price>
       </Order>
   </PurchaseLineItems>
</xyz:PurchaseOrder>
```

5.4.2.1 Partially Transmitted Arrays

SOAP provides support for partially transmitted arrays, known as "varying" arrays in some contexts [12]. A partially transmitted array indicates in an "SOAP-ENC:offset" attribute the zero-origin offset of the first element transmitted. If omitted, the offset is taken as zero.

The following is an example of an array of size five that transmits only the third and fourth element counting from zero:

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[5]" SOAP-
ENC:offset="[2]">
   <item>The third element</item>
   <item>The fourth element</item>
</SOAP-ENC:Array>
```

5.4.2.2 Sparse Arrays

SOAP provides support for sparse arrays. Each element representing a member value contains a "SOAP-ENC:position" attribute that indicates its position within the array. The following is an example of a sparse array of two-dimensional arrays of strings. The size is 4 but only position 2 is used:

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[,][4]">
   <SOAP-ENC:Array href="#array-1" SOAP-ENC:position="[2]"/>
</SOAP-ENC:Array>
<SOAP-ENC:Array id="array-1" SOAP-
ENC:arrayType="xsd:string[10,10]">
   <item SOAP-ENC:position="[2,2]">Third row, third col</item>
   <item SOAP-ENC:position="[7,2]">Eighth row, third col</item>
</SOAP-ENC:Array>
```

If the only reference to array-1 occurs in the enclosing array, this example could also have been encoded as follows:

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[,][4]">
  <SOAP-ENC:Array SOAP-ENC:position="[2]" SOAP-
ENC:arrayType="xsd:string[10,10]>
    <item SOAP-ENC:position="[2,2]">Third row, third col</item>
    <item SOAP-ENC:position="[7,2]">Eighth row, third
col</item>
  </SOAP-ENC:Array>
</SOAP-ENC:Array>
```

5.4.3 Generic Compound Types

The encoding rules just cited are not limited to those cases where the accessor names are known in advance. If accessor names are known only by inspection of the immediate values to be encoded, the same rules apply, namely that the accessor is encoded as an element whose name matches the name of the accessor, and the accessor either contains or references its value. Accessors containing values whose types cannot be determined in advance MUST always contain an appropriate xsi:type attribute giving the type of the value.

Similarly, the rules cited are sufficient to allow serialization of compound types having a mixture of accessors distinguished by name and accessors distinguished by both name and ordinal position. (That is, having some accessors repeated.) This does not require that any schema actually contain such types, but rather says that if a type-model schema does have such types, a corresponding XML syntactic schema and instance may be generated.

```
<xyz:PurchaseOrder>
   <CustomerName>Henry Ford</CustomerName>
   <ShipTo>
       <Street>5th Ave</Street>
       <City>New York</City>
       <State>NY</State>
       <Zip>10010</Zip>
   </ShipTo>
   <PurchaseLineItems>
       <Order>
           <Product>Apple</Product>
           <Price>1.56</Price>
       </Order>
       <Order>
           <Product>Peach</Product>
           <Price>1.48</Price>
       </Order>
   </PurchaseLineItems>
</xyz:PurchaseOrder>
```

Similarly, it is valid to serialize a compound value that structurally resembles an arrray but is not of type (or subtype) SOAP-ENC:Array. For example:

```
<PurchaseLineItems>
    <Order>
        <Product>Apple</Product>
        <Price>1.56</Price>
    </Order>
    <Order>
        <Product>Peach</Product>
        <Price>1.48</Price>
```

```
    </Order>
</PurchaseLineItems>
```

5.5 Default Values

An omitted accessor element implies either a default value or that no value is known. The specifics depend on the accessor, method, and its context. For example, an omitted accessor typically implies a Null value for polymorphic accessors (with the exact meaning of Null accessor-dependent). Likewise, an omitted Boolean accessor typically implies either a False value or that no value is known, and an omitted numeric accessor typically implies either that the value is zero or that no value is known.

5.6 SOAP root Attribute

The SOAP root attribute can be used to label serialization roots that are not true roots of an object graph so that the object graph can be deserialized. The attribute can have one of two values, either "1" or "0". True roots of an object graph have the implied attribute value of "1". Serialization roots that are not true roots can be labeled as serialization roots with an attribute value of "1" An element can explicitly be labeled as not being a serialization root with a value of "0".

The SOAP root attribute MAY appear on any subelement within the SOAP Header and SOAP Body elements. The attribute does not have a default value.

6. Using SOAP in HTTP

This section describes how to use SOAP within HTTP with or without using the HTTP Extension Framework. Binding SOAP to HTTP provides the advantage of being able to use the formalism and decentralized flexibility of SOAP with the rich feature set of HTTP. Carrying SOAP in HTTP does not mean that SOAP overrides existing semantics of HTTP but rather that the semantics of SOAP over HTTP maps naturally to HTTP semantics.

SOAP naturally follows the HTTP request/response message model providing SOAP request parameters in a HTTP request and SOAP response parameters in a HTTP response. Note, however, that SOAP intermediaries are NOT the same as HTTP intermediaries. That is, an HTTP intermediary addressed with the HTTP Connection header field cannot be expected to inspect or process the SOAP entity body carried in the HTTP request.

HTTP applications MUST use the media type "text/xml" according to RFC 2376 [3] when including SOAP entity bodies in HTTP messages.

6.1 SOAP HTTP Request

Although SOAP might be used in combination with a variety of HTTP request methods, this binding only defines SOAP within HTTP POST requests (see section 7 for how to use SOAP for RPC and section 6.3 for how to use the HTTP Extension Framework).

6.1.1 The SOAPAction HTTP Header Field

The SOAPAction HTTP request header field can be used to indicate the intent of the SOAP HTTP request. The value is a URI identifying the intent. SOAP places no restrictions on the format or specificity of the URI or that it is resolvable. An HTTP client MUST use this header field when issuing a SOAP HTTP Request.

```
soapaction    = "SOAPAction" ":" [ <"> URI-reference <"> ]
URI-reference = <as defined in RFC 2396 [4]>
```

The presence and content of the SOAPAction header field can be used by servers such as firewalls to appropriately filter SOAP request messages in HTTP. The header field value of empty string ("") means that the intent of the SOAP message is provided by the HTTP Request-URI. No value means that there is no indication of the intent of the message.

Examples:

```
SOAPAction: "http://electrocommerce.org/abc#MyMessage"
SOAPAction: "myapp.sdl"
SOAPAction: ""
SOAPAction:
```

6.2 SOAP HTTP Response

SOAP HTTP follows the semantics of the HTTP Status codes for communicating status information in HTTP. For example, a 2xx status code indicates that the client's request including the SOAP component was successfully received, understood, and accepted etc.

In case of a SOAP error while processing the request, the SOAP HTTP server MUST issue an HTTP 500 "Internal Server Error" response and include a SOAP message in the response containing a SOAP Fault element (see section 4.4) indicating the SOAP processing error.

6.3 The HTTP Extension Framework

A SOAP message MAY be used together with the HTTP Extension Framework [6] in order

to identify the presence and intent of a SOAP HTTP request.

Whether to use the Extension Framework or plain HTTP is a question of policy and capability of the communicating parties. Clients can force the use of the HTTP Extension Framework by using a mandatory extension declaration and the "M-" HTTP method name prefix. Servers can force the use of the HTTP Extension Framework by using the 510 "Not Extended" HTTP status code. That is, using one extra round trip, either party can detect the policy of the other party and act accordingly.

The extension identifier used to identify SOAP using the Extension Framework is

```
http://schemas.xmlsoap.org/soap/envelope/
```

6.4 SOAP HTTP Examples

Example 3 SOAP HTTP Using POST

```
POST /StockQuote HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<SOAP-ENV:Envelope...

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope...
```

Example 4 SOAP Using HTTP Extension Framework

```
M-POST /StockQuote HTTP/1.1
Man: "http://schemas.xmlsoap.org/soap/envelope/"; ns=NNNN
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
NNNN-SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<SOAP-ENV:Envelope...

HTTP/1.1 200 OK
Ext:
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
```

```
<SOAP-ENV:Envelope...
```

7. Using SOAP for RPC

One of the design goals of SOAP is to encapsulate and exchange RPC calls using the extensibility and flexibility of XML. This section defines a uniform representation of remote procedure calls and responses.

Although it is anticipated that this representation is likely to be used in combination with the encoding style defined in section 5 other representations are possible. The SOAP encodingStyle attribute (see section 4.3.2) can be used to indicate the encoding style of the method call and or the response using the representation described in this section.

Using SOAP for RPC is orthogonal to the SOAP protocol binding (see section 6). In the case of using HTTP as the protocol binding, an RPC call maps naturally to an HTTP request and an RPC response maps to an HTTP response. However, using SOAP for RPC is not limited to the HTTP protocol binding.

To make a method call, the following information is needed:

- The URI of the target object
- A method name
- An optional method signature
- The parameters to the method
- Optional header data

SOAP relies on the protocol binding to provide a mechanism for carrying the URI. For example, for HTTP the request URI indicates the resource that the invocation is being made against. Other than it be a valid URI, SOAP places no restriction on the form of an address (see [4] for more information on URIs).

7.1 RPC and SOAP Body

RPC method calls and responses are both carried in the SOAP Body element (see section 4.3) using the following representation:

- A method invocation is modelled as a struct.
- The method invocation is viewed as a single struct containing an accessor for each

[in] or [in/out] parameter. The struct is both named and typed identically to the method name.

- Each [in] or [in/out] parameter is viewed as an accessor, with a name corresponding to the name of the parameter and type corresponding to the type of the parameter. These appear in the same order as in the method signature.
- A method response is modelled as a struct.
- The method response is viewed as a single struct containing an accessor for the return value and each [out] or [in/out] parameter. The first accessor is the return value followed by the parameters in the same order as in the method signature.
- Each parameter accessor has a name corresponding to the name of the parameter and type corresponding to the type of the parameter. The name of the return value accessor is not significant. Likewise, the name of the struct is not significant. However, a convention is to name it after the method name with the string "Response" appended.
- A method fault is encoded using the SOAP Fault element (see section 4.4). If a protocol binding adds additional rules for fault expression, those also MUST be followed.

As noted above, method and response structs can be encoded according to the rules in section 5, or other encodings can be specified using the encodingStyle attribute (see section 4.1.1).

Applications MAY process requests with missing parameters but also MAY return a fault.

Because a result indicates success and a fault indicates failure, it is an error for the method response to contain both a result and a fault.

7.2 RPC and SOAP Header

Additional information relevant to the encoding of a method request but not part of the formal method signature MAY be expressed in the RPC encoding. If so, it MUST be expressed as a subelement of the SOAP Header element.

An example of the use of the header element is the passing of a transaction ID along with a message. Since the transaction ID is not part of the signature and is typically held in an infrastructure component rather than application code, there is no direct way to pass the necessary information with the call. By adding an entry to the headers and giving it a fixed name, the transaction manager on the receiving side can extract the transaction ID and use it without affecting the coding of remote procedure calls.

8. Security Considerations

Not described in this document are methods for integrity and privacy protection. Such issues will be addressed more fully in a future version(s) of this document.

9. References

[1] S. Bradner, "The Internet Standards Process -- Revision 3", RFC2026, Harvard University, October 1996

[2] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, Harvard University, March 1997

[3] E. Whitehead, M. Murata, "XML Media Types", RFC2376, UC Irvine, Fuji Xerox Info. Systems, July 1998

[4] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

[5] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, U.C. Irvine, DEC W3C/MIT, DEC, W3C/MIT, W3C/MIT, January 1997

[6] H. Nielsen, P. Leach, S. Lawrence, "An HTTP Extension Framework", RFC 2774, Microsoft, Microsoft, Agranat Systems

[7] W3C Recommendation "The XML Specification"

[8] W3C Recommendation "Namespaces in XML"

[9] W3C Working Draft "XML Linking Language". This is work in progress.

[10] W3C Working Draft "XML Schema Part 1: Structures". This is work in progress.

[11] W3C Working Draft "XML Schema Part 2: Datatypes". This is work in progress.

[12] Transfer Syntax NDR, in "DCE 1.1: Remote Procedure Call"

[13] N. Freed, N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC2045, Innosoft, First Virtual, November 1996

A. SOAP Envelope Examples

A.1 Sample Encoding of Call Requests

Example 5 Similar to Example 1 but with a Mandatory Header

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
       <t:Transaction
           xmlns:t="some-URI"
           SOAP-ENV:mustUnderstand="1">
               5
       </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DEF</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 6 Similar to Example 1 but with multiple request parameters

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceDetailed
```

```
          xmlns:m="Some-URI">
            <Symbol>DEF</Symbol>
            <Company>DEF Corp</Company>
            <Price>34.1</Price>
        </m:GetLastTradePriceDetailed>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A.2 Sample Encoding of Response

Example 7 Similar to Example 2 but with a Mandatory Header

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    <SOAP-ENV:Header>
        <t:Transaction
          xmlns:t="some-URI"
          xsi:type="xsd:int" mustUnderstand="1">
            5
        </t:Transaction>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <m:GetLastTradePriceResponse
          xmlns:m="Some-URI">
            <Price>34.5</Price>
        </m:GetLastTradePriceResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 8 Similar to Example 2 but with a Struct

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

```
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse
         xmlns:m="Some-URI">
           <PriceAndVolume>
               <LastTradePrice>
                   34.5
               </LastTradePrice>
               <DayVolume>
                   10000
               </DayVolume>
           </PriceAndVolume>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 9 Similar to Example 2 but Failing to honor Mandatory Header

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
       <SOAP-ENV:Fault>
           <faultcode>SOAP-ENV:MustUnderstand</faultcode>
           <faultstring>SOAP Must Understand
Error</faultstring>
       </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 10 Similar to Example 2 but Failing to handle Body

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
       <SOAP-ENV:Fault>
           <faultcode>SOAP-ENV:Server</faultcode>
           <faultstring>Server Error</faultstring>
           <detail>
```

```
                <e:myfaultdetails xmlns:e="Some-URI">
                  <message>
                    My application didn't work
                  </message>
                  <errorcode>
                    1001
                  </errorcode>
                </e:myfaultdetails>
            </detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
